EZ-Link平台开发手册

Draft lite V1.0

EZ-Link平台开发手册 1

1 平台简介 4

1.1 PCB设计 4

1.2 SDK架构 5

1.3 源码目录 8

2 开发环境 10

2.1 配置文件 10

2.2 库文件 11

2.3 IDE配置 12

2.4 编程框架 15

2.5 烧录Flash 17

2.6 调试 19

3 IOT接口 20

3.1 系统服务 20

3.2 硬件服务 21

3.3 WIFI 25

3.4 Bluetooth 27

3.5 GSM 31

3.6 GPS 34

缩略词：

|  |  |
| --- | --- |
| BCPU | Baseband CPU |
| BTCPU | Bluetooth CPU |
| XCPU | System CPU |
| BSS | Basic Service Set |
| BSSID | BSS ID |
| COS | Common Operating System |
|  |  |
|  |  |

# 平台简介

EZ-Link是RDA物联网应用集成开发平台，该平台包括SDK开发包，Eclipse集成开发环境，调试工具和烧录工具。SDK开发包提供了在EZ-Link平台进行物联网开发必需的模块和标准API接口，同时通过对硬件层抽象，提供对RDA物联网系列芯片的全面支持。

RDA物联网系列芯片即有独立蓝牙、WIFI芯片，也有在SoC上集成Wi-Fi、Bluetooth和GSM功能的组合芯片,并提供包括UART/ADC/I2S在内的硬件接口。

RDA特有的共存技术使用户可以独立或并行使用Wi-Fi及蓝牙功能。

本文阐述EZ-Link平台SDK的主要功能和使用方法。本文所描述SDK为精简版，该版本针对物联网设备开发周期短、市场反应敏感等特点，对EZ-Link SDK进行了一系列优化升级，以帮助客户提高开发效率。用户仅需要了解必要的接口，然后通过SDK提供的丰富示例代码，简单修改甚至零修改就可以实现一款完整的物联网应用。

## PCB设计

请参考各芯片数据手册。

## SDK架构

EZ-Link SDK具有模块化、可扩展的架构和清晰的层次结构，灵活的SDK架构为产品定制提供软件支撑，用户可以根据不同市场需求和变化，进行快速调整。

EZ-Link SDK的架构如下，每一部分将会在下面具体描述。

除了APP应用层需要客户完成，其余模块将由RDA统一打包发布给客户。

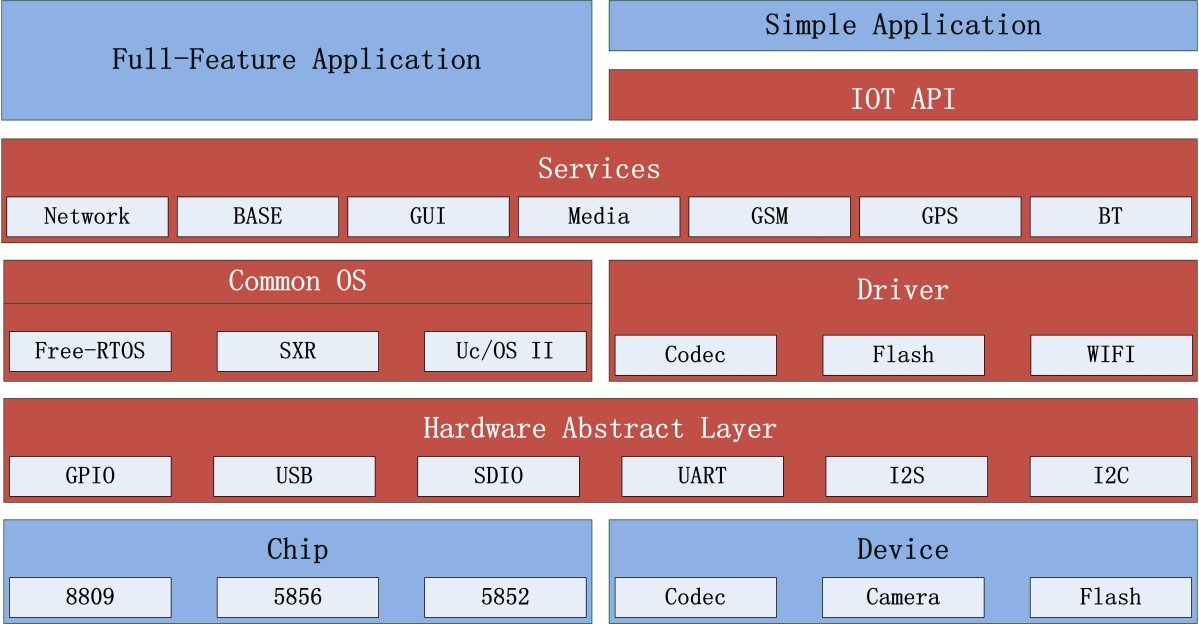


图1-1 EZ-Link SDK架构

|  |
| --- |
| **硬件层** |

最底层为硬件层，包括芯片和外设硬件。

|  |
| --- |
| **硬件抽象层** |

第二层为硬件抽象层，该层隐藏底层硬件细节，为上层驱动提供统一接口，为实现不同芯片间平滑切换提供软件支持。

|  |
| --- |
| **驱动和操作系统层** |

第三层为驱动和操作系统层。RDA在RTOS之上又提供了一层瘦COS层，该COS层封装了核心操作系统接口如任务调度，并发处理等，为服务层提供统一接口。COS层为用户提供两方面便利：用户可以随心所欲选择自己偏好的操作系统；用户可以移植已有的应用程序，复用代码，提高代码利用效率，加速开发进度。

|  |
| --- |
| **服务层** |

第四层为服务层，服务层应用程序类似windows服务程序或linux deamon程序，这些程序长时间执行，为上层用户和应用提供特定的服务，如http，GPS服务等。上层用户可以通过标准接口使用服务层程序提供的服务，如多媒体应用程序可以通过GUI接口显示播放界面，同时可以通过media服务进行音频数据解码和播放。服务层应用程序通常为操作系统的任务(task)，该任务直接调用COS接口以降低与RTOS的耦合。

服务层应用程序十分丰富，可以满足当前主流的物联网应用。良好的可扩展架构使用户可以添加更多自有服务程序或第三方应用程序，满足客户特殊的产品需求。

该层同时提供精简的IOT API，通过精简IOT API，用户可以快速开发一款简单的物联网应用，如智能水杯等。

|  |
| --- |
| **应用程序层** |

最高层为应用程序层，用户通常工作在本层。应用程序层通过调用服务层接口进行软件开发。

EZ-Link各层之间以松散的方式耦合，各层提供的接口和实现分离，层内关注实现，层间关注接口。除此之外，EZ-Link SDK同样遵循模块化设计原则，方便用户对产品特性进行裁剪以适应客户需求，发挥用户平台、硬件资源的最大效率:

|  |  |
| --- | --- |
| 芯片模块化 | RDA 芯片为本SDK设计参照的核心，芯片类型决定系统的CPU硬件架构、IO接口、总线等特性，进而影响服务层提供的服务、产品特性和源码编译等。芯片模块化使用户在无需修改上层软件的情况下，根据需求选择适当的芯片 |
| 操作系统模块化 | 目前市面上嵌入式实时操作系统种类繁多，用户可以根据自己的偏好和使用习惯选择操作系统而不影响其他应用 |
| 驱动模块化 | 用户通常有更换外设的需求，驱动模块化正是满足该项需求 |
| 服务模块化 | 服务层模块化，有效减少各个子模块之间耦合，方便用户删减与特定产品无关服务，添加新服务 |

|  |
| --- |
| **特性** |

* 硬件层

本SDK支持多种RDA芯片。丰富的IO接口方便连接各种外设

* HAL层

硬件抽象层为上层驱动提供统一接口，方便驱动程序移植

* 驱动模块

驱动模块包括：GPIO,、UART、flash、LCD等

* RTOS层

强大的实时操作系统：

* 支持多任务处理
* 定时器管理
* 支持并发访问控制锁和任务间同步信号量
* 任务间通信机制
* 动态内存管理和glibc兼容的内存操作接口
* COS层

提供对RTOS封装，可针对不同RTOS进行适配，使服务层和应用层程序摆脱对操作系统依赖

* 系统服务层

EZ-Link将系统功能抽象成模块化的服务，并提供标准的服务接口，极大简化编程过程。针对物联网特点，EZ-Link提供丰富的网络协议，方便不同设备商之间的智能设备互联互通：

* 支持AllSeen协议
* 支持lwIP协议栈和http等应用层协议，方便扩展互联网应用
* 多媒体功能，音频、视频等
* GPS定位服务
* 支持DLNA协议
* GSM/GPRS服务
* 硬件层服务，主要针对GPIO、UART等串行总线接口
* OS服务，主要封装定时器、内存操作、调试等接口
* IOT API

简易IOT接口，简化编程，快速生成IOT应用，实现简单IOT产品

* 全功能应用层

相对简单应用程序层，实现更复杂的物联网应用

## 源码目录

SDK源码目录结构如下，各个目录将会在下面详细描述。

|  |
| --- |
| **SDK源码目录结构** |

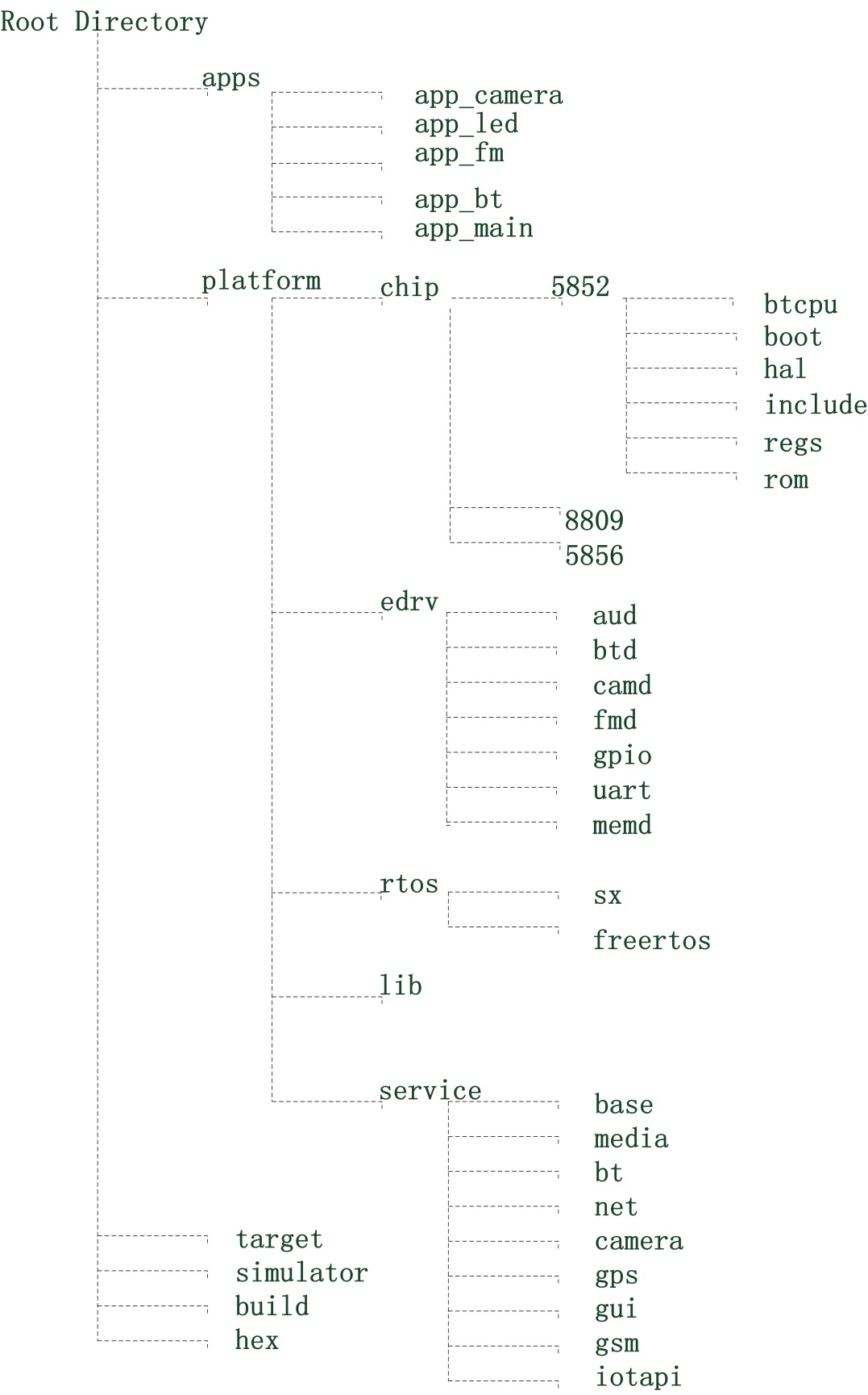


图1-2 EZ-Link SDK源码目录结构

图中各目录模块说明如下：

|  |  |  |
| --- | --- | --- |
| **目录** | **说明** | |
| apps | 客户应用程序和RDA示例代码 | |
| platform | RDA IOT平台代码，其下包括：chip、edrv、 rtos、 service等子目录，子目录会在下面依次说明 | |
| chip | 平台芯片相关代码，包括：boot loader，hal层，寄存器地址定义和rom代码。通常用户无需修改rom代码 | |
| edrv | 包括外设驱动和芯片内驱动，外设驱动包括：LCD，FM等；芯片内部驱动为内部总线驱动，如GPIO，UART等 | |
| rtos | 实时操作系统层，该目录包括原生操作系统 | |
| service | base | 基础服务，包括RTOS操作等 |
| net | 网络服务，提供http，lwna，tcp/ip操作接口 |
| bt | 蓝牙服务 |
| camera | 提供相机操作接口 |
| gps | 提供GPS定位服务 |
| gui | 用户界面操作接口 |
| gsm | GSM/GPRS操作服务接口 |
| media | 音频、视频服务接口等 |
| iotapi | 简单iot应用程序编程接口 |
| target | 用户产品配置文件 | |
| build | 影子目录，该目录在编译时由编译脚本生成，用来存放编译过程中生成的临时文件 | |
| lib | 模块静态连接库 | |
| hex | Lod文件、Ramrun文件和其他GDB相关asm及map文件 | |

表1-1 EZ-LINK SDK目录结构描述

# 开发环境

EZ-LINK选用windows操作系统作主机（HOST），在windows系统中搭建交叉编译环境。

## 配置文件

模块化的设计为用户定制产品提供了便利。通常定制产品通过修改配置文件来完成。配置文件在target目录，EZ-LINK配置文件主要涉及以下几个：

|  |  |
| --- | --- |
| **配置文件** | **说明** |
| target.def | 产品配置，target表示产品、开发板 |
| target\_common.def | 对target.def信息的集中处理，通常不需要用户修改 |
| memd.def | Flash属性定义 |
| tgt\_app\_cfg.h | 用户功能宏头文件，控制用户APP特性  文件存在./target/include |
| tgt\_board\_cfg.h | 硬件线路板配置  文件存在./target/include |
| tgt\_calib\_cfg.h | 校准配置  文件存在./target/include |

表2-1 配置文件

产品定义配置文件target.def包含下面一些选项，各选项精确的函数请参考配置文件说明注释：

|  |  |  |
| --- | --- | --- |
| **选项** | **说明** | |
| 芯片描述信息 | 定义芯片类型和封装 | |
| 触屏驱动 | 定义触屏设置 | |
| FM驱动 | 定义FM总线和时钟管脚 | |
| FLASH驱动 | 定义FLASH类型、时钟及ADMUX | |
| LCD驱动 | 定义LCD总线 | |
| CAMERA驱动 | 是否支持CAMERA及各种特性 | |
| 蓝牙驱动 | 定义蓝牙模块类型，32K时钟管脚，UART透传 | |
| T卡驱动 | 定义T卡接口类型 | |
| 音频驱动 | 定义音频设置 | |
| WIFI驱动 | 定义WIFI 32K时钟管脚，TCP/IP支持，SDIO，GPIO，SDMMC接口支持，DLNA接口支持，WIFI音箱支持 | |
| 32K时钟配置 | 配置时钟 | |
| 用户项目定义 | APP\_BASEDIR | 客户APP目录 |
| CT\_PROJECT | 定义资源类型 |
| AUD\_MAX\_LEVEL | 定义音量 |
| RESOURCE\_SUPPORT | 是否包含资源 |
| AT\_SUPPORT | 是否支持AT命令 |
| BT\_SPP | 是否支持蓝牙SPP |
| USB支持类型 | 是否支持USB音箱；USB HOST | |
| 编译优化选项 | CT\_OPT：优化编译空间  WITHOUT\_WERROR：告警是否停止编译 | |
| 调试选项 | TRACE控制选项； | |

表2-2 产品配置文件选项

## 库文件

EZ-Link开发平台库文件统一放置在./platform/lib/chip\_name目录。用户开发程序时引用的头文件放置在各模块./inlcude目录。

库文件分为release和debug两个版本，通过文件名区分。debug版本包含调试信息，并可以使用GDB工具进行单步调试，调试信息可以通过串口打印到控制台；release版本不包含调试信息。正式发布的产品bin文件应当链接release版本库以减少bin文件体积，提高运行速度。

## IDE配置

EZ-LINK SDK选择Eclipse作为集成开发环境，SDK编译工具链统一集成在Eclipse中。

|  |  |
| --- | --- |
| D:\doc\image\MainLCD\WarningPage\PopUpPage\PP_INF.gif | Eclipse和JRE（Jave Run Environment）选用32位版本，可以兼容64和32位操作系统 |

|  |
| --- |
| **安装Eclipse** |

1. 下载EZ-LINK集成开发工具包，其中包含JRE安装程序和Eclipe软件
2. 安装JRE，安装过程始终选择默认安装选项
3. Eclipse无需安装，直接解压运行

Eclipse成功运行说明集成开发环境工作正常，下一步通过Eclipse配置编译工具，导入SDK库文件和源码。

|  |
| --- |
| **创建工程** |

打开Eclipse, 选择菜单“File/New”, 然后选择“RDA Project”。项目配置窗口如下，自定义工程名，选择项目代码路径：

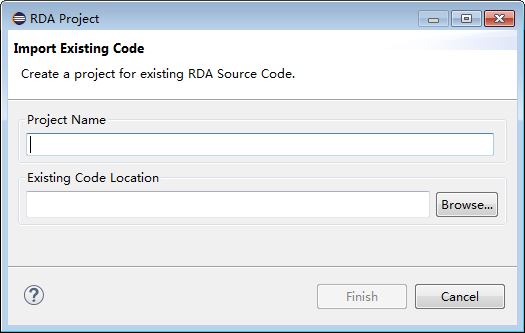


图2-1 创建项目

项目基础代码由RDA提供，用户可以在基础代码之上实现自己特有的功能和应用。

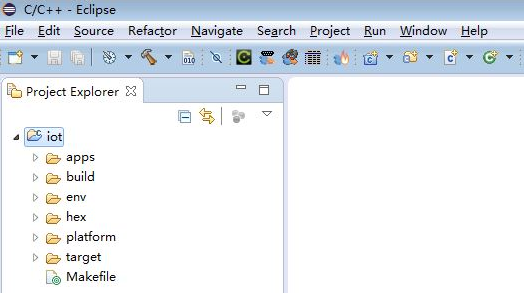
文档中示例工程名为“iot”，选择代码路径后单击“Finish”按钮，生成工程如下：

图2-2 项目浏览器

|  |
| --- |
| **配置工程** |

在“Project Explorer”中点击右键，选择“Properties”，进入“Properties”窗口，选择“C/C++ Build”，然后选择“RDA Project”，配置目标：

|  |  |
| --- | --- |
| Select Target | 选择目标Target |
| Target Wizard | 项目向导，配置项目特性、模块等 |
| Remove | 删除Target |
| New Target | 新建Target |
| Select Release | 选择生成“release”还是“debug”版本 |
| Base Build Command | - |
| Custom Build Option | - |

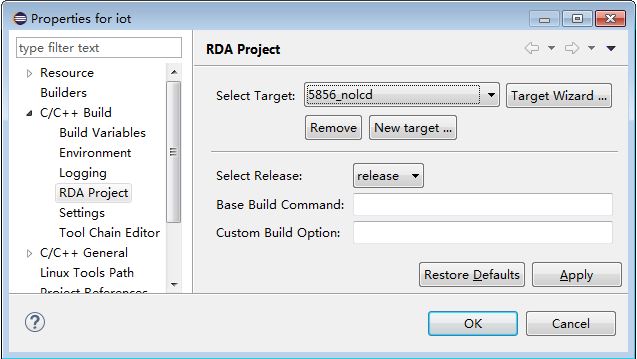


图2-3 项目配置

|  |
| --- |
| **编译工程** |

在“Project Explorer”中点击右键，选择“RDA Tools”，然后可以看到两个编译选项：

|  |  |
| --- | --- |
| Build Flash | 编译Ramrun文件 |
| Build Image | 编译Flash文件 |

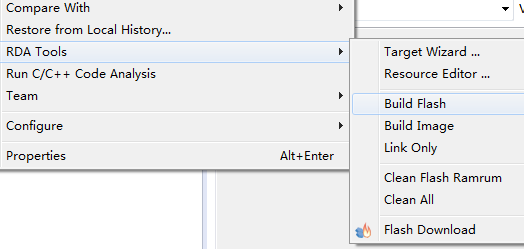


图2-4 编译项目

|  |
| --- |
| **编译模块** |

在“Project Explorer”中展开根目录 iot，选择模块，在模块上点击右键，选择“RDA Tools”，然后可以看到如下两个选项：

|  |  |
| --- | --- |
| Build this module | 编译该模块 |
| Clean this module | 清除模块已编译中间文件 |

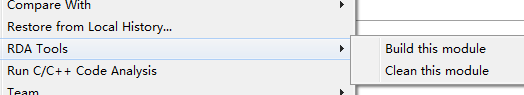


图2-5 编译模块

## 编程框架

对EZ-LINK开发平台而言，短短数十行代码就可以完成一个简单的物联网应用，如智能LED灯等。简单物联网应用编程框架有两部分组成：

1. init()函数，负责资源初始化
2. loop()函数，持续监听系统事件，并处理。loop函数不间断运行直至系统关机

|  |
| --- |
| **用户模块组织结构** |

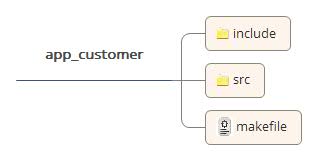


图2-6 用户代码结构

include目录存放用户头文件，如果用户无自定义头文件，该目录可以为空，或者直接删除。

src目录存放用户源文件。源文件中实现init()和loop()，用户在init()中实现初始化代码，在loop()中处理外设如WIFI模块产生的事件。源文件结构参考示例代码。

makefile文件具有固定格式，简单易用。模块makefile甚至无需修改就可以正确编译用户代码，将用户代码与系统库文件一起编译链接生成可烧录的flash二进制文件。

|  |
| --- |
| **Makefile格式** |

|  |
| --- |
| ## ----------------------------------------------------------- ##  ## Don't touch the next line unless you know what you're doing.##  ## ----------------------------------------------------------- ##  **include ${SOFT\_WORKDIR}/env/compilation/compilevars.mk**  # Name of the module  **LOCAL\_NAME := apps/ap\_customer**  # list all modules APIs that are neccessary to compile this module  **LOCAL\_API\_DEPENDS := \**  **platform \**  **platform/service \**  ## ----------------------------------------------------------- ##  ## List all your sources here ##  ## ----------------------------------------------------------- ##  **include src/make.srcs**  ## ----------------------------------------------------------- ##  ## Do Not touch below this line ##  ## ----------------------------------------------------------- ##  **include ${SOFT\_WORKDIR}/env/compilation/compilerules.mk** |

## 烧录Flash

|  |
| --- |
| **安装串口驱动** |

烧录BIN文件使用USB转串口线，RDA提供的开发包中包含USB转串口驱动程序，点击安装。驱动安装成功后，依次查看windows“控制面板”->“设备管理”->“端口”，在“端口”选项下，有如下所示USB-to-Serial端口，说明安装成功：

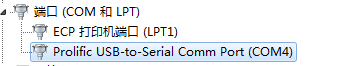


图2-7 安装串口驱动

|  |
| --- |
| **配置串口** |

在Eclipse菜单项选择“windows”，然后选择“Preference”，选择“C/C++”，然后选择“RDA Tools”。在“Serial Setting”中选择串口和正确的Baudrate。

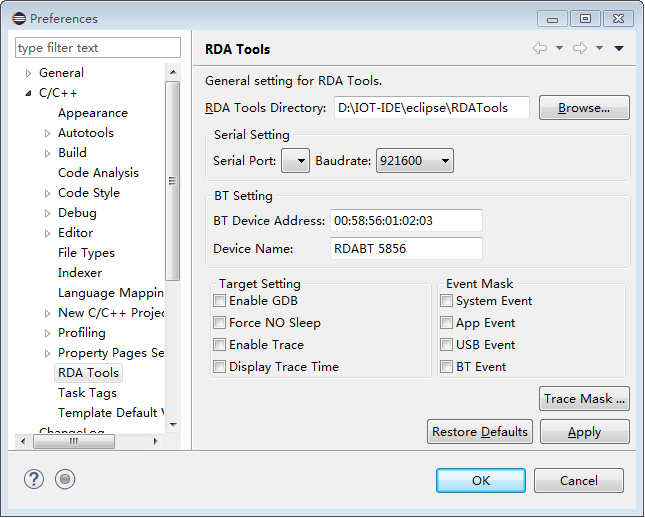


图2-8 串口配置

|  |
| --- |
| **Eclipse烧录工具条目** |

烧录相关按钮在Eclipse工具栏，如下图示。



自左向右分别是：

|  |  |
| --- | --- |
| Reset Target | 重置目标板，目标板将重启 |
| Connect Target | 连接目标板，成功失败都会有弹出对话框提示 |
| Open Coolwatcher | 打开CoolWatcher工具 |
| Open Memory View | 打开内存读写窗口，该部分将在2.6 调度中介绍 |
| Down Flash | 下载Ramrun 和Flash 文件 |

|  |
| --- |
| **烧录BIN文件** |

连接UART COM口：蓝线连接HST\_RXD，白线连接HST\_TXD，黑线接地。

启动Eclipse，点击工具栏“Connect Target”，连接成功后，点击“Down Flash”，选择Lod文件，点击“Download”：

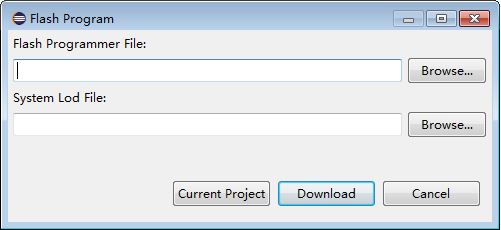


图2-9 下载选项

|  |  |
| --- | --- |
| D:\doc\image\MainLCD\WarningPage\PopUpPage\PP_INF.gif | Flash Programmer File文件由*build flash*产生  System Lod File文件由*build image*产生，参考2.3 IDE配置 |

## 调试

串口打印trace是最为常见的调试手段。EZ-LINK trace按模块分类，模块内按重要性分为不同等级。

通过宏开关，可以选择编译时是否包含特定模块trace。

通过Eclipse工具，可以选择运行时是否输出特定模块和该模块特定级别的trace。设置步骤如下：

在Eclipse菜单栏选择“windows”，然后选择“Preference”，选择“C/C++”，然后选择“RDA Tools”，最后单击“Trace Mask”按钮，弹出如下对话框。

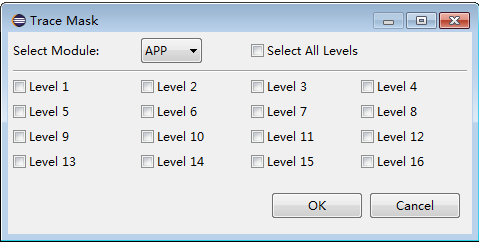


图2-10 配置Trace

“Select Module”选择trace模块，16个level控制trace输出级别。

# IOT接口

## 系统服务

### 内存

头文件：iot\_base.h

#### iot\_Malloc

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 分配内存, glibc兼容 | | | |
| 函数： | PVOID iot\_Malloc(UINT32 size) | | | |
| 参数： |  | | | |
| 【输入】 | size | 申请内存大小 | | |
| 返回值： | 指向分配内存的指针 | |  |  |

#### iot\_Free

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 释放内存, glibc兼容 | | | |
| 函数： | VOID iot\_Free(PVOID ptr) | | | |
| 参数： |  | | | |
| 【输入】 | ptr | 内存指针 | | |
| 返回值： | 无 | |  |  |

### Trace

头文件iot\_base.h

#### iot\_Printf

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 打印trace，示例打印LED灯状态： | | | |
|  | iotPrintf("led status = %d ", ledOn); | | | |
| 函数： | VOID iot\_Printf(INT8\* fmt,...) | | | |
| 参数： |  | | | |
| 【输入】 | fmt | 打印格式 | | |
| 返回值： | 无 | |  |  |

### 延时

#### iot\_DelayMs

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 程序延时 | | | |
| 函数： | VOID iot\_delayMs(UIN32 ms) | | | |
| 参数： |  | | | |
| 【输入】 | ms | 延时ms毫秒 | | |
| 返回值： | 无 | |  |  |

#### iot\_DelaySeconds

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 程序延时 | | | |
| 函数： | VOID iot\_DelaySeconds( UINT32 seconds) | | | |
| 参数： |  | | | |
| 【输入】 | secondes | 定时器时长，以秒为单位 | | |
| 返回值： | 无 | |  |  |

## 硬件服务

### UART

#### uart\_Open

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 打开UART接口 | | | |
| 函数： | VOID uart\_Open(UINT8 id, UINT32 baudrate) | | | |
| 参数： |  | | | |
| 【输入】 | id | Uart ID | | |
| 【输入】 | baudrate | Uart 波特率 | | |
| 返回值： | 无 | |  |  |

#### uart\_Close

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 关闭UART接口 | | | |
| 函数： | VOID uart\_Close(UINT8 id); | | | |
| 参数： |  | | | |
| 【输入】 | id |  | | |
| 返回值： | 无 | |  |  |

#### uart\_Write

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 通过UART接口发送数据，每次只发送一个字节数据 | | | |
| 函数： | UINT32 uart\_Write(UINT8 id, CHAR ch); | | | |
| 参数： |  | | | |
| 【输入】 | uartId | UART ID | | |
| 【输入】 | ch | 数据地址指针 | | |
| 返回值： | > 0 | | - | 通过UART发送字节数 |
|  | 0 | | - | 发送数据失败 |

#### uart\_Read

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 通过UART接口读数据 | | | |
| 函数： | UINT32 uart\_Read(UINT8 id, CHAR\* buff) | | | |
| 参数： |  | | | |
| 【输入】 | id | UART ID | | |
| 【输入】 | buff | 数据地址指针 | | |
| 返回值： | > 0 | | - | 通过UART接收字节数 |
|  | 0 | | - | 接收数据失败 |

### GPIO

#### gpio\_Open

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 打开GPIO接口 | | | |
| 函数： | VOID gpio\_Open(UINT8 port, UINT8 direction) | | | |
| 参数： |  | | | |
| 【输入】 | port | 端口号 | | |
| 【输入】 | direction | 0 – 输入 ；1 – 输出 | | |
| 返回值： | 无 | |  |  |

#### gpio\_Close

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 关闭GPIO接口 | | | |
| 函数： | VOID gpio\_Close(UINT8 port) | | | |
| 参数： |  | | | |
| 【输入】 | port | 端口号 | | |
| 返回值： | 无 | |  |  |

#### gpio\_Write

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 向GPIO写“0”或者“1” | | | |
| 函数： | VOID gpio\_Write(UINT8 port, UINT8 data) | | | |
| 参数： |  | | | |
| 【输入】 | port | 端口号 | | |
| 【输入】 | data | 数据 | | |
| 返回值： | 无 | |  |  |

#### gpio\_Read

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 从GPIO读数据 | | | |
| 函数： | UINT32 gpio\_Read(UINT8 port) | | | |
| 参数： |  | | | |
| 【输入】 | port | 端口号 | | |
| 返回值： | GPIO接口当前数据 | |  |  |

#### 示例程序

简单物联网应用：智能LED灯，通过WIFI无线控制LED灯开关。现实生活中，可以减少装修布线，减轻对墙体破坏等。

|  |
| --- |
| // app\_led.c  #include “hal\_gpio.h”  #include “tcpip\_sockets.h”  static struct sockaddr\_in ledSckAddr;  static INT8 ledSckServer;  static INT8 ledSckClient = 0;  ///Only accept 1 byte data(0 or 1), ok for turn on/off the LED  #define MAX\_RECV\_BUFF 1  #define LED\_SCK\_PORT 5050  #define MAX\_LED\_CONNECTION 1  VOID init(VOID)  {  struct sockaddr\_in clientAddr;  HAL\_GPIO\_GPIO\_ID\_T ledGpioId;  HAL\_GPIO\_DIRECTION\_T direction;  ///Use the tenth GPIO port  ledGpioId = HAL\_GPIO\_10;  direction = HAL\_GPIO\_DIRECTION\_OUTPUT;  ///Must open the GPIO before use it  **gpio\_Open(ledGpioId, direction );**  ledSckServer = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);  memset(&ledSckAddr,0, sizeof(struct sockaddr\_in));    ledSckAddr.sin\_family=AF\_INET;  ledSckAddr.sin\_addr.s\_addr=htonl(INADDR\_ANY);  ledSckAddr.sin\_port=htons(LED\_SCK\_PORT);  if(-1 == bind(ledSckServer,  (struct sockaddr \*)(& ledSckAddr), sizeof(sockaddr\_in)))  {  return ;  }  }  VOID loop(VOID)  {  INT8 readLen = 0;  UINT8 ledOn;  /// If no connection, waiting for connect  if (ledSckClient <= 0)  {  if(-1 == listen(ledSckServer, MAX\_LED\_CONNECTION))  {  return ;  }  addrSize = sizeof(struct sockaddr\_in);  ledSckClient = accept(ledSckServer,  (struct sockaddr \*)(& clientAddr), &addrSize);  if(-1 == ledSckClient)  {  return ;  }  }  readLen = read(ledSckClient, &ledOn, MAX\_REV\_BUF\_SIZE);  if (readLen > 0)  {  if (0 == ledOn || 1 == ledOn)  **gpio\_Write(ledGpioId, ledOn);**  }  } |

## WIFI

### 编程接口

头文件：iot\_wifi.h

#### iot\_Connect

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 连接网络 | | | |
| 函数： | INT32 iot\_Connect(const char \*addr, INT16 portno) | | | |
| 参数： |  | | | |
| 【输入】 | addr | 网络地址，URL或者 IP | | |
| 【输入】 | portno | 端口号 | | |
| 返回值： | < 0 | |  | 失败 |
|  | 其他 | |  | 成功 |

#### iot\_Send

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 发送数据 | | | |
| 函数： | INT32 iot\_Send(INT32 sockfd, const INT8\* buffer, UINT32 len) | | | |
| 参数： |  | | | |
| 【输入】 | sockfd | 套接字句柄 | | |
| 【输入】 | buffer | 发送数据缓存 | | |
| 【输入】 | len | 发送数据长度 | | |
| 返回值： | 指定索引的BSSID字串 | |  |  |

#### iot\_Recv

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 接收数据 | | | |
| 函数： | INT32 iot\_Recv(INT32 sockfd, INT8 \*buffer, UINT32 len) | | | |
| 参数： |  | | | |
| 【输入】 | sockfd | 套接字句柄 | | |
| 【输入】 | buffer | 接收数据缓存 | | |
| 【输入】 | len | 接收数据长度 | | |
| 返回值： | 实际接收字节数 | |  |  |

#### iot\_Close

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 关闭网络 | | | |
| 函数： | INT32 iot\_Close(INT32 sockfd) | | | |
| 参数： |  | | | |
| 【输入】 | sockefd | 套接字句柄 | | |
| 返回值： |  | |  |  |

### 示例程序

|  |
| --- |
| #include "iot\_wifi.h"  // Source device id allocated by oneNet  #define SRC\_DEV "777"  // Destination device id allocated by oneNet  #define DST\_DEV "778"  // API key allocated by oneNet  #define API\_KEY "your api key"  // Data stream name, created on oneNet  #define DATA\_STREAM "your data stream"  #define MAX\_RESP\_BUF 32  // ============================================================================  // Loop forever  // Send temperature value to cloud application whenever changed  // @return void  // ============================================================================  VOID loop()  {  // Server information for connection  INT8 \*server\_ip = "183.230.40.39";  INT8 \*port = "876";  INT8 \*src\_dev = SRC\_DEV;  INT8 \*dst\_dev = DST\_DEV;  INT8 \*src\_api\_key = API\_KEY;  INT8 \*data\_stream = DATA\_STREAM;  DOUBLE temperature = 0.0;  INT sock, ret;  EdpPacket \*pkg = NULL;  INT8 buffer[MAX\_RESP\_BUF] = { 0 };  SaveDataType data\_type = kTypeSimpleJsonWithoutTime;    recv\_buf = NewBuffer();  // Connect to server  sock = iot\_Connect(server\_ip, atoi(port));  if (sock < 0)  {  iot\_Printf("Error connect cloud server. \n");  return;  }  // Build connect package follow oneNet EDP protocol  pkg = PacketConnect1(src\_dev, src\_api\_key);  if (pkg == NULL)  {  iot\_Printf("Build connect pkg failed.\n");  return;  }  // Send connection request pkg to cloud server  ret = iot\_Send(sock, (const char\*)pkg->\_data, pkg->\_write\_pos);  DeleteBuffer(&pkg);  while (1)  {  Temperature = user\_poll\_current\_temperature();  // Build save data pkg follow oneNet EDP protocol  pkg = PacketSavedataDouble(data\_type,dst\_dev,data\_stream,temperature,0, NULL);  if (pkg == NULL)  {  iot\_Printf("Build data failed.\n");  return;  }  // Send temperature value pkg to clould server  iot\_Send(sock, (const char\*)pkg->\_data, pkg->\_write\_pos);  DeleteBuffer(&pkg);  iot\_DelayMs(2000);  }  // Close socket  iot\_Close(sock);  return;  } |

## Bluetooth

### 编程接口

头文件：iot\_bt.h

#### bt\_Open

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 启动蓝牙模块 | | | |
| 函数： | VOID bt\_Open(VOID) | | | |
| 参数： |  | | | |
| 【输入】 | 无 |  | | |
| 返回值： | 无 | |  |  |

#### bt\_Close

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 关闭蓝牙模块 | | | |
| 函数： | VOID bt\_Close(VOID) | | | |
| 参数： |  | | | |
| 【输入】 | 无 |  | | |
| 返回值： | 无 | |  |  |

#### bt\_SetLocalName

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 设置本地蓝牙设备名 | | | |
| 函数： | VOID bt\_SetLocalName(UINT8\* name) | | | |
| 参数： |  | | | |
| 【输入】 | name | 本地蓝牙设备名 | | |
| 返回值： | 无 | | - |  |

#### bt\_SetVisible

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 设置本地蓝牙可见性 | | | |
| 函数： | VOID bt\_SetVisable(UINT8 visable); | | | |
| 参数： |  | | | |
| 【输入】 | visible | TRUE – 可见；FALSE – 不可见 | | |
| 返回值： | 无 | | - |  |

#### bt\_ScanDevice

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 扫描周边蓝牙设备 | | | |
| 函数： | UINT32 bt\_ScanDevice(UINT32 timeout) | | | |
| 参数： |  | | | |
| 【输入】 | timeout | 扫描超时时间，单位秒 | | |
| 返回值： | 获取蓝牙设备总数 | |  |  |

#### bt\_BoneDevice

|  |  |  |  |
| --- | --- | --- | --- |
| 功能描述： | 配对蓝牙设备 | | |
| 函数： | UINT32 bt\_BondDevice( t\_bdaddr device\_addr, UINT32 timeout ) | | |
| 参数： |  | | |
| 【输入】 | Timeout | 配对超时时间，单位秒 | |
| 返回值： | TRUE 配对成功  FALSE 配对失败 | |

#### bt\_GetBonedDevice

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 蓝牙配对 | | | |
| 函数： | BOOL bt\_GetBonedDevice(UINT32 index, rdabt\_device\_t\_app \*device); | | | |
| 参数： |  | | | |
| 【输入】 | Index | 蓝牙设备索引，取值范围0~9 | | |
| 【输出】 | device | 蓝牙设备信息数据指针 | | |
| 返回值： | TRUE | | - | 获取成功 |
|  | FALSE | | - | 获取失败 |

#### bt\_SppConnect

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 连接蓝牙SPP服务端 | | | |
| 函数： | BOOL bt\_SppConnect(t\_bdaddr device\_addr, UINT8 timeout); | | | |
| 参数： |  | | | |
| 【输入】 | device\_addr | 蓝牙地址 | | |
| 【输入】 | timeout | 蓝牙连接超时 | | |
| 返回值： | TRUE | | - | 连接成功 |
|  | FALSE | | - | 连接失败 |

#### bt\_SppDisconnect

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 断开蓝牙SPP连接 | | | |
| 函数： | VOID bt\_SppDisconnect(VOID) | | | |
| 参数： |  | | | |
| 【输入】 | 无 |  | | |
| 返回值： | 无 | |  |  |

#### bt\_SppSend

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 通过SPP发送数据 | | | |
| 函数： | INT32 bt\_SppSend(UINT8\* buf, UINT32 size) | | | |
| 参数： |  | | | |
| 【输入】 | buf | 发送数据缓冲区 | | |
| 【输入】 | size | 发送数据长度 | | |
| 返回值： | > 0 | | - | 实际发送字节数 |
|  | <= 0 | | - | 发送失败 |

#### bt\_SppRecv

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 通过蓝牙SPP接收数据 | | | |
| 函数： | INT32 bt\_SppRecv(UINT32 timeout， UINT8 \*buf) | | | |
| 参数： |  | | | |
| 【输入】 | timeout | 超时时间，单位秒 | | |
| 【输出】 | buf | 接收数据缓冲区 | | |
| 返回值： | > 0 | | - | 成功接收数据字节数 |
|  | <= 0 | | - | 接收数据失败 |

#### bt\_SppAccept

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 蓝牙SPP服务端接受客户端连接 | | | |
| 函数： | BOOL bt\_SppAccept(UINT32 timeout) | | | |
| 参数： |  | | | |
| 【输入】 | timeout | 超时时间，单位秒 | | |
| 返回值： | TRUE | | - | 连接成功 |
|  | FALSE | | - | 连接失败 |

### 示例程序

简单物联网应用：智能插座，通过蓝牙无线控制插座的开关。

|  |
| --- |
| // app\_smart\_plug.c  #include “bt\_types.h”  #include “hal\_gpio.h”  static HAL\_GPIO\_GPIO\_ID\_T plugGpioId;  VOID init(VOID)  {  HAL\_GPIO\_DIRECTION\_T direction;  // Use the fifth GPIO port  plugGpioId = HAL\_GPIO\_15;  direction = HAL\_GPIO\_DIRECTION\_OUTPUT;  // Must open the GPIO before use it  gpio\_Open(plugGpioId, direction );  **bt\_Open(VOID);**  }  VOID loop(VOID)  {  t\_DataBuf btData;    memcpy(&btData, 0, sizeof(t\_DataBuf));  // Return only when data available or error  **bt\_SppRecv(&btData);**  if (btData.len > 0)  {  if (0 == (\*btData.buff) || 1 == (\*btData.buff) )  {  gpio\_Write(plugGpioId, \*btData.buff);  }  }  } |

## GSM

### 编程接口

#### gsm\_xxx

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： |  | | | |
| 函数： |  | | | |
| 参数： |  | | | |
| 【输入】 |  |  | | |
| 【输入】 |  |  | | |
| 返回值： |  | | - |  |

#### gsm\_xxx

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： |  | | | |
| 函数： |  | | | |
| 参数： |  | | | |
| 【输入】 |  |  | | |
| 返回值： |  | | - |  |

#### gsm\_xxx

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： |  | | | |
| 函数： |  | | | |
| 参数： |  | | | |
| 【输入】 |  |  | | |
| 【输入】 |  |  | | |
| 返回值： |  | | - |  |

#### gsm\_xxx

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： |  | | | |
| 函数： |  | | | |
| 参数： |  | | | |
| 【输入】 |  |  | | |
| 【输入】 |  |  | | |
| 返回值： |  | | - |  |

#### gsm\_xxx

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： |  | | | |
| 函数： |  | | | |
| 参数： |  | | | |
| 【输入】 |  |  | | |
| 【输入】 |  |  | | |
| 返回值： |  | | - |  |

#### gsm\_xxx

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： |  | | | |
| 函数： |  | | | |
| 参数： |  | | | |
| 【输入】 |  |  | | |
| 【输入】 |  |  | | |
| 返回值： |  | | - |  |

#### gsm\_xxx

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： |  | | | |
| 函数： |  | | | |
| 参数： |  | | | |
| 【输入】 |  |  | | |
| 【输入】 |  |  | | |
| 返回值： |  | | - |  |

#### gsm\_xxx

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： |  | | | |
| 函数： |  | | | |
| 参数： |  | | | |
| 【输入】 |  |  | | |
| 【输入】 |  |  | | |
| 返回值： |  | | - |  |

#### gsm\_xxx

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： |  | | | |
| 函数： |  | | | |
| 参数： |  | | | |
| 【输入】 |  |  | | |
| 【输入】 |  |  | | |
| 返回值： |  | | - |  |

#### gsm\_xxx

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： |  | | | |
| 函数： |  | | | |
| 参数： |  | | | |
| 【输入】 |  |  | | |
| 【输入】 |  |  | | |
| 返回值： |  | | - |  |

### 示例程序

## GPS

### 编程接口

#### gps\_Open

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 打开GPS模块 | | | |
| 函数： |  | | | |
| 参数： | VOID gps\_Open(VOID) | | | |
| 【输入】 | 无 |  | | |
| 返回值： | 无 | | - |  |

#### gps\_Close

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 关闭GPS模块 | | | |
| 函数： | VOID gps\_Close(VOID) | | | |
| 参数： |  | | | |
| 【输入】 | 无 |  | | |
| 返回值： | 无 | | - |  |

#### gps\_GetData

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： | 获取当前GPS信息 | | | |
| 函数： | gps\_GetData | | | |
| 参数： |  | | | |
| 【输入】 |  |  | | |
| 【输入】 |  |  | | |
| 返回值： |  | | - |  |

#### gps\_xxx

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能描述： |  | | | |
| 函数： |  | | | |
| 参数： |  | | | |
| 【输入】 |  |  | | |
| 【输入】 |  |  | | |
| 返回值： |  | | - |  |

### 编程示例